Analysis of software vulnerability detection methods*

Liudmyla Gryzun[1,†], Alla Havrylova[2,†,*], Andrii Tkachov[2,†], Andrii Hapon[3,†] and Natalia Brynza[1,†]

[1] Simon Kuznets Kharkiv National University of Economics, Nauky Av.,9A, Kharkiv, Ukraine
[2] National Technical University "Kharkiv Polytechnic University", Kyrpychova Str., 2, Kharkiv, Ukraine
[3] Kharkiv National University of Radio Electronics, Nauky Av.,14, Kharkiv, Ukraine

Abstract

The author proposes a direction of improvement of existing software protection systems, focusing efforts on increasing their ability to detect new types of malware. The most promising direction for the development of technologies for detecting vulnerabilities and vulnerabilities in software is an approach that combines different methods of analysis. This allows achieving higher accuracy of the results, as well as increasing the productivity of the tools used to check the code. A method of combining static analysis of programs and dynamic symbolic execution is proposed to improve the accuracy of vulnerability detection while maintaining high performance of analysis tools. This approach will significantly reduce the risk of errors that can be missed when using one of the analysis methods separately, and also improves the efficiency of the overall software security process.

Keywords

malware, antivirus solution, analysis, cybersecurity, method

1. Introduction

Despite great efforts, both theoretical and practical, to solve the problem of information security, its current state is far from any reliable solution. Moreover, a number of new problems have arisen that are related to protection against malware. Experts are even radically changing their views on the problem of organising cyber defence in view of the threats posed by malware. malware makes it impossible for a computer to function properly, so new anti-virus measures and intrusion detection systems are undoubtedly needed to treat programs, destroy viruses and prevent them. A large number of different methods are currently used to solve this problem.

These approaches have both strengths and weaknesses. According to experts in the field, methods that use the pattern approach have become very effective. To ensure the smooth operation of pattern scanners in anti-virus programs and intrusion detection systems, it is necessary to constantly maintain the databases that contain malware samples.

2. Problematic issues of software vulnerability detection

The essence of malware detection is the identification of patterns or signs that are characteristic of malware. This may include analysing the programme code for suspicious instructions or behaviour that is not typical of legitimate programmes.

There can be several reasons for undetected malware [1–3].

Proceedings of the Workshop on Scientific and Practical Issues of Cybersecurity and Information Technology at the V international scientific and practical conference Information security and information technology (ISecIT 2025), June 09–11, 2025, Lutsk, Ukraine
* Corresponding author.
† These authors contributed equally.
Liudmyla.Gryzun@hneu.net (L. Gryzun); sharaya1972@gmail.com (A. Havrylova); andrii.tkachov@khpi.edu.ua (A. Tkachov); andrii.hapon@nure.ua (A. Hapon), natalia.brynza@hneu.net (N. Brynza)
0000-0002-5274-5624 (L. Gryzun), 0000-0002-2015-8927 (A. Havrylova); 0000-0003-1428-0173 (A. Tkachov); 0000-0003-2560-7426 (A. Hapon),), 0000-0002-0229-2874 (N. Brynza)

1. Volume – although the exact number is unknown, researchers are finding hundreds of thousands of new malware every day. The sheer number of malware means that some of the newest malware can bypass common cybersecurity measures to protect a corporate information system.

2. Outdated antivirus software – outdated antivirus software that blocks malware by detecting patterns relies on an updated database to stop the latest known threats.

3. Undetectable malware - Some malware can be difficult to detect due to its sophisticated design, allowing it to bypass cybersecurity mechanisms. For example, some malware may be specifically designed to trick popular anti-virus software into believing it is harmless. Other malware may use social engineering to trick users into installing it.

The use of intrusion detection and anti-virus software that is constantly updated is an important element of the defence against new and evolving threats in the field of software protection.

Based on the analysis of known approaches to detecting malware in software code, the following methods of malware detection can be distinguished, which in turn are based on statistical or dynamic tracking of code activity.

Static analysis methods involve checking code without executing it to identify malicious patterns or known malware, while dynamic analysis methods execute code in a controlled environment to observe its behaviour and detect suspicious activity. Dynamic analysis may include elements of using algorithms to determine the likelihood that a program is malicious based on its characteristics and behaviour (heuristic analysis) and monitoring the program to detect abnormal behaviour that may indicate malicious intent (behavioural analysis).

One way malware spreads is by copying itself into every executable file it infects. These types of malware replicate an identical copy of themselves, byte by byte, each time they infect a new file. They can be easily detected by looking for a specific string of bytes, called a ‘pattern’, that is derived from the malware body. The latter may include:

- character string;

- semantic expressions in a special language;

- formal mathematical model, etc.

Patterns are identified by experts in the field of computer virology. They are able to isolate the virus code from the program code and formulate its characteristic properties in the most searchable form. Almost every company that develops anti-virus software has its own group of specialists who analyse new viruses and add new patterns to the anti-virus database.

The algorithm of the pattern-based method is based on the search for patterns – attacks in the source data collected by network and host sensors of the attack detection system. When patterns are detected, the attack detection system records the fact of an information attack that matches the found signature.

The essence of a standard pattern-based virus detection strategy is to maintain a database of malicious patterns. Each incoming file is checked for a virus signature that matches an entry in the database of known patterns. In this case, the protection system provides for the constant updating of the pattern database and relies on it.

Pattern-based malware detection is the most common method used by commercial antiviruses. But despite its simplicity, it has the following significant drawbacks.

1. Protection is provided only against known malware. It should be noted here that patterns are usually created to cover as many viruses as possible – a virus family. However, there is always a change in an executable file that can cause the pattern to stop being detected.

2. Constant growth of the pattern database. As the number of viruses, their types, and the ability of viruses to change increases, the speed of filling the database also increases.

3. When a virus appears and before the pattern database is updated, the client is vulnerable to a new malware. Only by identifying the file under investigation as a virus can its pattern be retrieved and added to the database. Moreover, malware developers have learnt to successfully bypass pattern detection by obfuscating the virus body. This has forced anti-virus companies to develop alternative protection methods.

With the anomaly detection approach, the developers of an anti-virus solution build a database of actions that are considered safe. If an application's execution process violates any of these predefined rules, it is marked as malicious. While anomaly-based detection has the potential to detect new malware patterns, it also has a very high false positive rate. Many anti-virus packages use algorithms for analysing the sequence of commands to generate some statistics and make decisions about the possibility of infection for each object being scanned. They are called heuristic scanning methods. Moreover, unlike the pattern-based method, the heuristic approach can detect both known and unknown viruses (i.e. those created after heuristic processing). In this approach, analysts use machine learning techniques to classify malware. Static, dynamic, visual representations of features, or a combination of these are used to train a classifier on a dataset consisting of both malicious and non-malicious binary files.

Various machine learning techniques such as support vector machine (SVM), random forests (RF), decision trees (DT), naive bayes classifier (NB), k-nearest neighbours (kNN) and gradient boosting are used to classify and detect malware samples and their respective classes, to filter out malware that requires further investigation by an analyst. Static analysis is the process of analysing binary malware without actually running the code. Patterns found during this analysis include pattern strings, frequency distribution of byte sequences or opcodes, byte-level n-grams or opcode-level n-grams, API (Application Programming Interface) calls, the structure of a disassembled program, etc.

The following are studies that used static malware analysis approaches [2].

In [4], information was extracted from executable files, such as the list of DLLs (Dynamic Link Library) used inside the executable file; the list of DLL system calls; the number of different system calls inside each DLL; characters or strings encoded in a binary file using a hexadecimal dump as features. For classification, NB was used with a training dataset consisting of 4266 files, including 3265 viruses and 1001 clean samples. According to the authors, the result was an accuracy of 97.11%. This is one of the first attempts to analyse malware using data mining techniques.

In [5], the classification accuracy of various machine learning methods, such as NB, SVM, DT and their enhanced versions, was investigated for classifying malware in different families using the features proposed in [6]. DT (C4.5 implementation) proved to be the best, with 99.6 per cent accuracy and 2.7 per cent false positives.

In [7], they presented a new approach of using a variable-length instruction sequence to detect worms in binary files using machine learning. They used RF and DT algorithms to classify a dataset consisting of 1444 worms and 1330 clean files and achieved classification accuracy of 96 %.

In [8], they investigated the use of API call sequences for malware detection. They showed that all versions of the same malware have a common basic pattern that can be identified using a basic call sequence API.

In their study, [9] proposed a scheme based on the obfuscation technique to study the shortcomings of static analysis approaches. Experiments have shown that the static approach is not sufficient for effective malware analysis. Static analysis can be easily avoided if the malware is obfuscated or compressed. Therefore, you need to pay attention to the following behavioural features for better analysis. Dynamic analysis. Common to all Dynamic Approach approaches is the execution of a malware sample in a controlled environment to extract behavioural features (virtual machine, emulator, sandbox, etc.). The behaviour is monitored using tools such as Process Monitor, Process Explorer, Wireshark, or Capture BAT.

Study [10] proposed a comprehensive approach to conducting behaviour-based malware analysis and classifying malware into new groups using artificial intelligence methods.

They used a resource that is a bait for attackers – honeypots – and intrusion detection systems, such as HoneyClients and Amun, to collect malware samples. Next, a behavioural report was created for each sample using virtual machine platforms such as CWSandbox and Anubis, and each report was analysed manually. Using artificial intelligence methods, the malware samples were categorised into groups – worms and trojans. The main drawback of the study is that the analysis of the reports was not automated. Therefore, given the huge volume of malware being generated today, it is impossible to analyse reports manually.

In [11], they proposed a method for automated identification of new classes of malware with similar behaviour (clustering) and classification of previously unseen malware for the identified classes (classification) using machine learning. By using clustering and classification, a new approach is used to handle the behaviour of a large number of malware binaries. This approach significantly reduced the execution time of the analysis methods. The researchers captured state change features such as file opening, mutex blocking, network activity, infection of running processes, or registry key setting, and then mapped the malware behaviour in a multidimensional vector space. More than 10 000 malware samples belonging to 14 different families were used in the experiment. These malware samples were collected using honeypots and spam traps. The result was an accuracy of 88% in classifying the families. The disadvantage of the work is that only one binary execution path was considered in the analysis. It is clear from the above work that static or dynamic analysis alone is not sufficient to accurately and effectively classify malware samples. This is because the individual use of these methods can be easily circumvented by using code obfuscation or various runtime stopping techniques. Also, dynamic analysis cannot examine all the execution paths of a program file. The controlled environment in which malware is monitored is different from the real one, the program may behave differently because some malware behaviour can only be triggered under certain conditions, for example, with a specific command or on a specific system date and, as a result, cannot be detected in a virtual environment.

In [12], a new approach to training a malware classifier was proposed using both statistical and dynamic methods. As a result of training the model, it was found that statistical and dynamic analysis together work better than separately. The experiments were conducted using two different datasets and many machine learning algorithms, namely DT, kNN, Bayesian network and SVM.

In [13], a hybrid model for classifying binary files into clean and malicious ones was also proposed, which integrates both dynamic and static analysis functions. For this purpose, static information such as 'printableString' (a type of string with limited characters in ASN notation) and function length frequency were extracted, and dynamic information such as API parameters and API function names were extracted. To test the model, 2939 malicious and 541 clean samples were used. Then, using integrated meta-classifiers such as SVM, DT, and RF, the malware samples were classified with an accuracy of 97,055 %.

Another area is visualisation-based approaches to malware analysis.

Several tools were available for visualising and editing a binary file, which display the file in hexadecimal and ASCII formats but do not convey any structural information to the analyst.

In [14], a dot plot data visualisation technique was applied and it was shown that visualisation can be useful for identifying software design patterns – a technique for visualising software product patterns that provides a visual overview of the system structure. This technique is useful for designing software systems through sequential abstraction, a design pattern that helps eliminate redundancy. The sequence is broken down into tokens, and then the tokens are plotted on the graph as points.

In [15], they tried to detect and visualise viruses embedded inside an executable file using self-organised Kohonen maps with artificial neural networks, which is used to visualise multidimensional data [16], without using pattern information. The study found that each virus family has its own mask.

In [17], the use of graphical byte visualisation for automatic malware classification was first investigated. In the study, the entire malware sample was converted into a grey-scale image.

The research dataset consisted of 9 458 malware samples belonging to 25 different classes collected from the Anubis system [18]. A kNN model was trained on these images; the Euclidean distance was used to estimate the distance. The malware samples were grouped into the respective classes and the accuracy of 97,18 %.

In [19], they also converted the malware into a two-dimensional grey-scale image and classified the samples based on the resulting textures. They extracted common features based on the textures using the Gabor wavelet transform and the image descriptor GIST. The experimental dataset consisted of 3131 binary samples from 24 unique malware families. After building the feature vector, classification is performed on the malware samples using SVM machine learning techniques. As a result, the following accuracy was obtained 96,35 %.

In [20], a step-by-step approach was proposed to automatically classify malware into different families and detect new malware. The study used a combination of byte-to-grayscale image conversion, n-gram operating code, and an import function. The decision module uses these features to classify malware samples into their respective families and to identify new unknown malware. To detect new malware families, we applied the Shared Nearest Neighbour (SNN) algorithm. The model was trained on a database consisting of 21 740 malware samples from 9 different families. The result was a classification accuracy of 98.9 % and a detection rate of 86.7 %. The paper studies pattern-based and heuristic methods of malware detection. A separate analysis is devoted to the use of machine learning methods for malware classification. Various machine learning techniques for classifying and detecting malware samples and their respective classes, as well as filtering them, are investigated. The usefulness of graphical visualisation of bytes for detecting software design patterns for further automation of virus detection is shown. A comparative characterisation of modern, mainly heuristic, malware detection methods is made and

systematised by search accuracy values. Among the many existing approaches to malware analysis, dynamic analysis is the most appropriate, as it allows detecting the destructive activities of various programs directly at runtime.

Dynamic malware analysis identifies methods that can detect new malware to a certain extent due to the adaptation property [21, 22]. In addition, due to the growing requirements for security systems [21, 22], the comparison of malware detection methods should be based on the properties of verifiability and the ability to detect new malware: the presence of unknown unique characteristics, the level of false positives, and the blurring of the data under study.

The results of the analysis of methods for detecting new malware according to the defined criteria are presented in [1, 21, 22].

Since no effective methodology for detecting unknown malware has been identified at the moment, in order to effectively search for and destroy malware, additional research is required, taking into account all the features of using modern methods of malware detection, to develop a more effective method of protecting software from malware in modern corporate systems.

Based on the analysis of methods for detecting new malware, it can be said that for existing methods, due to their properties, detection of one of the most dangerous types of malware capable of metaprogramming its own code in order to implement a method of hiding from security systems is an extremely difficult task of introducing a large entropy of the characteristics of the samples under study to solve the classification problem. This type of virus includes polymorphic (oligomorphic) and metamorphic types of malware. Thus, each of the classes of existing methods (knowledge-based, artificial intelligence, machine learning, behavioural) is able to solve this problem to a certain extent, but under certain limitations.

The results of the comparative analysis according to the described criteria show that among them it is possible to distinguish the methods that have shown the most complete compliance with them – methods based on the synergy of several techniques for detecting and detecting malware. Thus, formalisation of imprecise knowledge and implementation of approximate reasoning in the field of malware detection allows detecting its destructive activity in conditions of certain vagueness of information about the state of the information system with the possibility of adaptation to detect similar types of malware [22].

3. Improvement of existing systems

Today, there are two strategies for detecting new malware [21, 22]: detection of anomalous activity based on the audit of functioning scenarios, detection based on the experience gained in combating known malware. To effectively detect new malware, it is advisable to use a hybrid approach based on combining the most effective methods of static and dynamic software analysis in order to protect it from the destructive effects of malware, since the number of new instances of it, characterised by a fundamentally new set of features, is quite small.

A priority area for improving existing software protection systems is to supplement the existing functionality with a method based on the theory of fuzzy logic. Moreover, the application of the chosen method should provide for the determination of the polymorphic (metamorphic) component of malware for each known type of malware, which will ensure the effective detection of new malware based on the identification of polymorphic (metamorphic) structures of existing viruses in the conditions of some inaccuracy (blurring) of information about the state of the information system.

Official data from software developers often does not confirm the declared level of accuracy of detecting new malware in practice, which indicates that the antivirus systems in question were tested under specific conditions that are too different from the real world.

Today, the issue of software security is of both practical and research interest. Software security is directly related to the search for errors and vulnerabilities.

Software vulnerabilities are potential entry points for malware and are directly related to software security. Vulnerability detection is a complex task that requires the direct involvement of an analyst. The difficulties lie not only in finding the state of the application in which the vulnerability manifests itself, but also in assessing what security aspects it can affect in these exploits of a given malware vulnerability [23].

Thus, it can be said that the task of malware identification can be indirectly interpreted as the task of finding software defects as potential entry points for malware, and blocking and controlling these defects as a function of the software security system. The size and complexity of software are constantly increasing. The size of the source text of modern programs and software systems can reach hundreds of millions of lines. This complexity leads to the fact that the creation of high-quality software becomes virtually impossible without the use of automatic means of checking programs for compliance with functional requirements for the software and the absence of software errors.

The presence of vulnerabilities in software is primarily due to the fact that the improvement of programmer training does not keep pace with the increase in the complexity and size of programs. Despite the development of software development methods and tools to support the software development process, released programs contain errors that can lead to incorrect operation of the program, unauthorized access to critical data, and execution of malicious code. That is, software errors should be considered in the context of potential entry points for malicious software.

According to the CVE Details portal (www.cvedetails.com), which provides statistics on registered vulnerabilities in programs based on information from the MITRE corporation since 1999, several thousand critical vulnerabilities are registered annually in programs used [24]. A reliable way to demonstrate vulnerability is to run a program on a set of input data, in which the vulnerability leads to specified violations, such as execution of code specified by the attacker or calling a library function with parameters controlled by the attacker. Software defects that underlie such vulnerabilities are called exploitable defects.

Currently, methods are being developed that allow generating some types of test sets automatically. Their development is actively engaged in by researchers whose work is based on the technologies of automatic generation of test sets and symbolic interpretation. Symbolic interpretation defines the process of converting input data in a program in the form of formulas over symbolic variables and constants. Input data for the program act as symbolic variables that can take on arbitrary values.

4. Software vulnerabilities and known approaches to their detection

As previously defined, software vulnerability is directly related to errors, which are potential entry points for destructive malware and operational burdens for cybersecurity quartermasters.

The possibility of a software error was first mentioned in 1842 by Ada Augusta Lovelace in her work "An Essay on the Analytical Engine, Introduced by Charles Babbage", the main narrative of which was that the analytical process should be uniformly carried out according to the necessary control data provided by the analytical engine, and this could also be a source of possible error. The

mechanism was believed to be error-free in its processes, but the cards (with control data) could give erroneous commands [25].

Thus, we can formulate the definition of a loophole in a program as:

- program defect: Any error made during the design or implementation of a program that, if not corrected, could cause the program to be vulnerable;

- application vulnerability: A flaw in an application that can be exploited to implement information security threats.

It is obvious that the concept of program insufficiency is defined through the concept of vulnerabilities in the design or implementation of the program [26]. In this case, it is possible to define the concept of the process of examination of the program source code, which consists in identifying program deficiencies (potentially vulnerable structures) in the program source code. The IEEE 1044-2009 "Standard Classification of Software Anomalies" provides a more detailed classification of various program anomalies (deviations from the norm):

- defect – an imperfection or deficiency in the operation of a product, in which the product does not meet requirements or specifications and requires correction or replacement;

- error – a human action that led to an incorrect result;

- failure – cessation of the ability of a product to perform a required function or the inability to perform it before certain limitations;

- fault – a message about an error in the program.

In 1993, the National Institute of Standards and Technology (National Institute of Standards and Technology) issued a manual entitled "Program Error Analysis", which provides the following definitions:

- anomaly – any condition that differs from what is expected;

- defect – any non-conformity for use or non-conformity to specification;

- error:

a) the difference between a calculated, observed, or measured value and a true, specified, or theoretically correct value or state;

b) incorrect step, process, or data definition;

c) incorrect result;

d) human action that led to an incorrect result;

- fault – an incorrect step, process, or data definition in a computer program (see also error);

- failure – the difference between the external result of program execution and the requirement for the software product.

It should be borne in mind that in reality there is no difference between the concepts of "defect", "error" and "failure", since these terms are interpreted differently in the community (even in the standards that use these terms).

For example, there are such definitions::

- error – a place in the source code of a program that can cause the program to crash or output incorrect output data on certain external data;

- defect – a place indicating a lack of source code that will not necessarily lead to incorrect operation of the program, but may worsen its performance characteristics (for example, a memory leak);

- vulnerability – an error in a program that can be exploited by an attacker to intentionally crash the program, execute arbitrary code, leak confidential data, or otherwise breach system security.

Due to the fact that there is no single interpretation of such terms as “defect”, “error”, “failure”, in this work the terms “error” and “defect” will be used as synonyms of the term “potential vulnerability”, and the term “failure” will be used to denote the state of an emergency termination of the program. The term “program vulnerability” will be used in the meaning of obtaining unauthorized access to data processed by the program, or in the meaning of unauthorized execution of program code, as an entry point for intercepting control by malicious software. In turn, the “Dictionary of the Ukrainian Language” provides the following definition of the term “vulnerability”: Vulnerability (vulnerability) is “the inability (for example, of a system or its part) to withstand the effects of an unfriendly environment; the degree of sensitivity to damage, damage; a weak point in one of the elements of the object of protection; a factor in the realization of a threat”.

In publications, “vulnerability” is defined as “weaknesses in source code that could potentially be exploited to cause loss or damage” [27];

The National Institute of Standards and Technology in NISTIR-8011 Volume 4 “Supporting Automation for Security Control Assessments. Software Vulnerability Management” defines “vulnerability” as a weakness in computational logic (e.g., code) found in software or hardware components that, if exploited, results in a negative impact on confidentiality, integrity, or availability.

Vulnerability mitigation in this context typically involves code changes, but can also include changes to specifications or even the discontinuation of specifications (e.g., complete removal of affected protocols or functionality).

According to the CVE Details (Common Vulnerabilities and Exposures) website, a database of common information security vulnerabilities, more than 94 000 critical vulnerabilities have been registered in programs released to the market.

According to statistics [28], by types of malicious impact on the program, the following vulnerabilities are most frequently registered:

- malicious code execution;

- program crash;

- overflow;

- execution of malicious code on the client side;

- obtaining unauthorized access to data (obtaining information);

- implementation in the query SQL.

Many software errors that lead to incorrect program behavior during execution can be divided into the following classes according to the types of harmful effects [29]:

- vulnerabilities that lead to data corruption during processing: integer overflow, data corruption in RAM, accessing an uninitialized memory block, accessing memory by an uninitialized or hanging pointer, data falsification, etc.;

- vulnerabilities that lead to unauthorized access to user data: obtaining unauthorized access to a database, obtaining unauthorized access to information in the RAM or permanent memory of a computing device, obtaining elevated data access privileges, etc.;

- vulnerabilities that lead to exhaustion of system resources such as heap memory, files, sockets, etc.;

- vulnerabilities that lead to a crash of the program execution: access to a memory area that does not belong to the program, division by zero, etc.;

- vulnerabilities that lead to malicious code execution: interception of the malicious code control path, execution of malicious code on the client side, implementation of a command in the command line, and others.

Statistics regarding the relationship between programming errors and vulnerabilities show that the majority of attacks on computer systems are related to programming errors. Here are some key statistics that demonstrate this connection [30]:

1. The relationship between bugs and vulnerabilities. NIST (National Institute of Standards and Technology) reports that about 70% of all software vulnerabilities are the result of code errors. This includes types of errors such as incorrect input validation, memory management, and logic errors.

According to WhiteHat Security, 55% of software vulnerabilities can be traced to insufficient error handling or improper exception handling, making this type of error particularly critical for security [31].

2. Common types of vulnerabilities related to errors in code. According to OWASP (Open Web Application Security Project), most of the critical vulnerabilities that make it onto their "OWASP Top 10" list are a direct result of programming errors. For example [32]:

- SQL Injection (SQLi): abuse due to insufficient validation or sanitization of input data (logical error).

- Cross-Site Scripting (XSS): A vulnerability that occurs due to poor input handling (processing of input data).

- Buffer Overflow: Exploitation by attackers of memory management errors, which are often the result of resource misallocation.

- The CWE/SANS Top 25 Most Dangerous Software Errors regularly notes that most of the most common and dangerous errors are related to memory management and input data.

3. Vulnerability Exploitation Statistics. According to Veracode research, 83% of applications have at least one vulnerability in their source code [30]. Synopsys, in its 2021 State of Open Source Security Report, found that 84% of commercial applications contain vulnerabilities related to memory management or input processing [33].

4. Common programming errors that lead to vulnerabilities [34]:

- buffer overflows and other memory management errors account for about 15% of all vulnerabilities in systems using the C/C++ language, according to data from MITRE;

- incorrect login verification led to 37% of web application attacks, according to the Akamai State of the Internet Security Report.

5. Vulnerability detection and remediation rate. Veracode State of Software Security 2021 notes that more than 70% of vulnerabilities discovered in software were related to errors made by developers during the coding stage [35].

When testing open source software for vulnerabilities, Sonatype in 2022 found that 85% of vulnerabilities in them arose due to errors in data processing logic or thread management [36].

6. Vulnerabilities related to memory management and exceptions. According to Microsoft Security Intelligence, vulnerabilities arising from incorrect memory management (and as a result, the possibility of buffer overflow attacks) are responsible for 70% of critical vulnerabilities in their software [35, 36].

Statistics clearly indicate a close relationship between software bugs and vulnerabilities. The majority of cyber incidents are related to logic errors, incorrect memory management, or insufficient input processing.

Effective testing and code quality management can significantly reduce the number of vulnerabilities, as most attacks exploit these bugs.

Conclusions

A direction for improving existing software protection systems is proposed, focusing efforts on increasing their ability to detect new types of malicious software. It is important that the chosen method involves determining the evolutionary (metamorphic) component of malicious software for each known class, which will allow detecting new malicious software based on identifying the evolutionary structures of existing viruses, even under conditions of a certain inaccuracy (fuzziness) of information about the state of information systems..

Based on the review of modern software analysis methods, the most promising direction for the development of vulnerability detection technologies and vulnerabilities in programs is an approach that combines various analysis methods.

This approach allows you to achieve higher accuracy of results, as well as increase the performance of tools used to verify the code.

Traditional methods, such as static and dynamic analysis, have their strengths and weaknesses: static analysis provides quick detection of possible vulnerabilities without executing the program, but has a high probability of false positives; while dynamic analysis executes the code in a real environment, allowing you to detect vulnerabilities that appear only during execution, but is much less productive due to the need to run the program.

In this regard, modern researchers focus on combined methods that can take into account the advantages of both approaches to minimize their disadvantages. One of the most promising approaches is hybrid program analysis, which includes the integration of static analysis with dynamic symbolic execution.

This combination allows not only to identify possible vulnerabilities in the code before it is run, but also to verify these results during the actual execution of the program.

This approach significantly reduces the number of false positives and increases the accuracy of critical error detection, since static analysis allows you to quickly identify possible dangers, and dynamic symbolic execution clarifies their reality in specific scenarios.

A method of combining static program analysis and dynamic symbolic execution is proposed to improve the accuracy of vulnerability detection while maintaining high performance of analysis tools.

This approach significantly reduces the risk of errors that may be missed when using one of the analysis methods separately, and also improves the efficiency of the overall process of ensuring software security.

Hybrid analysis also provides the ability to scale for large projects, where it is necessary to analyze a large amount of code quickly, but without losing accuracy. Special attention should be paid to aspects of optimizing computing resources during hybrid analysis, which is an important factor for applying this approach in real software production conditions.

Declaration on Generative AI

The authors have not employed any Generative AI tools.

References

[1] О. С. Савенко Критерії класифікації методів виявлення шкідливого програмного забезпечення. *Вісник Хмельницького національного університету*. № 1. С. 23-27. (2018). URL: https://journals.khnu.km.ua/ vestnik/pdf/tech/pdfbase/2018/2018_1/jrn/pdf/6.pdf.

[2] І. І. Жульковська, А. В. Плужник, О. А. Жульковський. Сучасні методи виявлення шкідливих програм. *Математичне моделювання.* № 1. С. 46-54. (2021). URL: http://nbuv.gov.ua/UJRN/Mm_2021_1_8.

[3] С. М. Лисенко, Р. В. Щука. Аналіз методів шкідливого програмного забезпечення в комп'ютерних системах. *Вісник Хмельницького національного університету*. № 2 (283). С. 101-107. (2020). doi: 10.31891/2307-5732-2020-283-2-101-107.

[4] M. G. Schultz, E. Eskin, F. Zadok. Data Mining Methods for Detection of New Malicious Executables. Proceedings of the *IEEE Symposium on Security & Privacy,* California, 14-16 May 2001, 38-49 (2001). URL: https:// doi.org/10.1109/secpri.2001.924286

[5] I. Santos, J. Devesa, F. Brezo, J. Nieves, P. G. Bringas, Semi-supervised Learning for Unknown Malware Detection. *Advances in Intelligent Systems and Computin.* vol. 91. pp 415–422. (2012). https://www.springer.com/ series/4240.

[6] Static and Symbolic Analysis. URL: https://www.talkcrypto.org/blog /2019/03/ 15/static-and-symbolic-analysis/ (дата звер-нення 16.04.2024).

[7] J. Z. Kolter, M.A. Maloof, Learning to Detect Malicious Executables in the Wild, *2004 International Conference on Knowledge Discovery and Data Mining*, Seattle, WA, USA (22-25 August), pp. 470-478. URL: https://doi.org/ 10.1145/1014052.1014105.

[8] A. Sung, J. Xu, P. Chavez, S. Mukkamala, Static Analyzer of Vicious Executables (SAVE). In Proc. of the *20th Annual Computer Security Applications,* (2004). doi: 10.1109/CSAC.2004. 37.

[9] A. Moser, C. Kruegel, E. Kirda. Limits of Static Analysis for Malware Detection. In IEEE Computer Society, (2007). doi: 10.1109/ACSAC.2007.4413008.

[10] E. Peter, T. Schiller. A practical guide to honeypots, URL: http://www.cs.wustl.edu/ ~jain/cse571-09/ftp/honey.pdf. (2008).

[11] Д. Комашинский, И. Котенко, Интеллектуальный анализ данных для выявления вредоносных программ. International Journal of Computing. 12(1), 63-74. (2014). URL: https://doi.org/10.47839/ ijc.12.1.589.

[12] K. Rieck, T. Holz, C. Willems, P. Dussel, P. Laskov. Learning and classification of Malware behaviour. *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, Springer, Berlin, Heidelberg. vol. 5137. pp 108–125. (2008). https://doi.org/10.1007/978-3-540-70542-0_6.

[13] A. Gerasimov, L. Kruglov. Reachability confirmation of statically detected defects using dynamic analysis. *Proceedings of the 11th International Conference on Computer Science and Information Technologies (CIST)*. Yerevan, (2017). URL: https://csit.am/2017/ Proceedings/DT/DT3.pdf.

[14] Y. A. Byeong, Y. J. Hyuk, K. Seoyeon, K. Taeguen. Malware Detection Using Dual Siamese Network Model. *Computer Modeling in Engineering & Sciences*. 141(1): 563-584. (2024). URL: https://doi.org/10.32604/cmes.2024.052403

[15] R. Islam, R. Tian, L. M. Batten, S. Versteeg, Classification of malware based on integrated static and dynamic features. *Journal of Network and Computer Applications*. Vol. 36, Is. 2, P. 646-656. *(*2013). URL: https://doi.org/10.1016/ j.jnca.2012.10.004.

[16] S. Rami, O. Khairuddin, A. Khairul, A. Zainol. A Survey on Malware Analysis Techniques: Static, Dynamic, Hybrid and Memory Analysis. *International Journal on Advanced Science, Engineering and Information Technology*, vol. 8, no. 4-2, pp. 1662-1671. (2018). doi: 10.18517/ijaseit.8.4-2.6827.

[17] J. Helfman, Dotplot patterns: A literal look at pattern languages. *TAPOS*. 2:31–41. (1995). URL: https://citeseerx.ist.psu.edu/document? repid=rep1&type=pdf&doi=9cb62eed1ee485813eac8df87485f89289b73197.

[18] CERT/CC, Carnegie Mellon University. URL: http: //www.cert.org/present/cert-overview-trends/module-4.pdf, May 2003.

[19] I. S. Yoo. Visualizing windows executable virus using self-organizing maps. In Proc. of *ACM workshop on Visualization and data mining for computer security*, p. 82 – 89 (2004). URL: https://doi.org/10.1145/1029208.102922.

[20] T. Kohonen, Self-Organizing Maps. Springer Series in Information Sciences (SSINF, vol. 30), (1995). URL: https://www.springer.com/ series/0710.

[21] P. Godefroid, M. Y. Levin, D. Molnar. Automated Whitebox Fuzz Testing. *NDSS'2008 Proceedings of the Network and Distributed Systems Security*, pp. 151-166, San Diego, February 8 - 11, (2008). URL: https://www.researchgate.net/publication/221655409_Automated_Whitebox_Fuzz_Testing.

[22] В. В. Фесьоха, Д. Ю. Кисиленко, О. М. Нестеров. Аналіз спроможності існуючих систем антивірусного захисту та покладених у їхню основу методів до виявлення нового шкідливого програмного забезпечення у військових інформаційних системах. Системи і технології зв'язку, інформатизації та кібербезпеки. Вип. № 3. С.143-156. (2023). doi: https://doi.org/10.58254/viti.3.2023.16. 143.

[23] O. Barabash, V. Sobchuk, A. Sobchuk, A. Musienko, O. Laptiev. Algorithms for synthesis of functionally stable wireless sensor network. *Advanced Information Systems*. 9(1), P. 70–79. (2025) https://doi.org/10.20998/2522-9052. 2025.1.08.

[24] A. Tkachov, A. Hapon, D. Balagura, O. Sievierinov, I. Bukatych, A. Havrylova. Analysis of the Software Security Protection. *2024 8th International Symposium on Multidisciplinary Studies and Innovative Technologies (ISMSIT), 07-09 November 2024*. Ankara, Turkiye, p. 1–8, 2024. doi: 10.1109/ISMSIT63511.2024. 10757202.

[25] C. Flanagan, K. Rustan M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, R. Stata. Extended static checking for Java. PLDI '02 Proceedings of the ACM SIGPLAN 2002 conference on Programming language design and implementation, pp. 234-245, Berlin, Germany – June 17 – 19. (2002). doi: 10.1145 /512529.512558/

[26] N. Lukova-Chuiko, O. Herasymenko, S. Toliupa, S. Laptiev, T. Laptieva, O. Laptiev. The method detection of radio signals by estimating the parameters signals of eversible Gaussian propagation. *2021 IEEE 3rd International Conference on Advanced Trends in Information Theory, ATIT 2021 – Proceedings*. (2021). P. 67–70. doi: 10.1109/ATIT54053.2021.9678856.

[27] E. J. Schwartz, Th. Avgerinos, D. Brumley. All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but might have been afraid to ask). *SP '10 Proceedings of the 2010 IEEE Symposium on Security and Privacy*, pp. 317-331, Oakland, CA, USA, May 16 – 19, 2010. doi: 10.1109/SP.2010.26.

[28] CVEdetails.com – powered by Security Scorecard. https://www.cvedetails.com.

[29] L. Liu, W. Bao-sheng, Y. Bo and Z. Qiu-xi. Automatic Malware Classification and NewMalware Detection using Machine Learning. *In Frontiers of Information Technology and Electronic Engineering*, 18(9):1336-1347. (2016). doi: 10.1631/FITEE.1601325.

[30] І. Субач, В. Фесьоха, Н. О. Фесьоха. Аналіз існуючих рішень запобігання вторгненням в інформаційно-телекомунікаційні мережі. *Information Technology and Security*. Vol. 5, № 1. С. 29-41. (2017). URL: http://nbuv. gov.ua/UJRN/inftech_ 2017_5_1_6.

[31] E. Iannone, R. Guadagni, F. Ferrucci, A. De Lucia and F. Palomba, The Secret Life of Software Vulnerabilities: A Large-Scale Empirical Study. *in IEEE Transactions on Software Engineering,* vol. 49, no. 1, pp. 44-63, 1 Jan. (2023). doi: 10.1109/TSE.2022.3140868.

[32] P. Anderson. The use and limitations of static-analysis tools to improve software quality. CrossTalk, *The Journal of Defense Software Engineering*, vol. 21, No. 6, pp. 18-21. (2008). URL: https://www.researchgate.net/publication/215835966_The_Use_and_Limitations_of_Static-Analysis_Tools_to_Improve_Software_Quali ty/stats.

[33] A. Turing. On Computable Numbers With an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, Volume s2-42, Issue 1, pp. 230–265. URL: https://doi.org/10.1112/plms/s2-42.1.230.

[34] Ch. Chen, B. Cui, J. Ma, R. Wu, J. Guo, W. Liu. A systematic review of fuzzing techniques. *Copmuters & Security*, Vol. 75, pp. 118-137. (2018). URL: https://doi.org/10.1016/j.cose. 2018.02.002.

[35] P. Godefroid, M. Y. Levin, D. Molnar, Automated Whitebox Fuzz Testing. *NDSS'2008 Proceedings of the Network and Distributed Systems Security*, pp. 151-166, San Diego, February 8 – 11. (2008). https://www.researchgate.net/ publication/22 1655409_Automated_Whitebox_Fuzz_Testing.

[36] E. J. Schwartz, Th. Avgerinos, D. Brumley. All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but might have been afraid to ask). *SP '10 Proceedings of the 2010 IEEE Symposium on Security and Privacy*, pp. 317-331, Oakland, CA, USA, May 16 – 19, 2010. doi: 10.1109/SP.2010.26.

[37] T. Wang, T. Wei, G. Gu, W. Zou. TaintScope: a Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection. *SP'10 123 Proceedings of the 2010 IEEE Symposium on Security and Privacy*, pp. 497- 512, Oakland, CA, USA, May 16 – 19. (2010). doi: 10.1109/SP.2010.37.

[38] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, G. Vigna: Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *23rd Annual Network and Distributed System Security Symposium, NDSS 2016* (*23rd Annual Network and Distributed System Security Symposium*, NDSS 2016). URL: https://doi.org/ 10.14722/ ndss.2016.23368.

[39] S. Yevseiev, Y. Khokhlachova, S. Ostapov, O. Laptiev et al. Models of socio-cyber-physical systems security. *Monographs, PC TECHNOLOGY CENTER*, number 978-617-7319-72-5.redif, December. (2023). doi: https://doi.org/ 10.15587/978-617-7319-72-5.